Managed Agreement: Generalizing Two Fundamental Distributed Agreement Problems

Emmanuelle Anceaume, Roy Friedman, Maria Gradinariu

I R I S A
INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES

PUBLICATION
INTERNE
N° 1785

MANAGED AGREEMENT: GENERALIZING TWO FUNDAMENTAL DISTRIBUTED AGREEMENT PROBLEMS

E. ANCEAUME R. FRIEDMAN* M. GRADINARIU
* COMPUTER SCIENCE DEPARTMENT, THE TECHNION, HAIFA

ISSN 1166-8687

IRISA
CAMPUS UNIVERSITAIRE DE BEAULIEU - 35042 RENNES CEDEX - FRANCE

INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES
Campus de Beaulieu – 35042 Rennes Cedex – France
Tél. : (33) 02 99 84 71 00 – Fax : (33) 02 99 84 71 71
http://www.irisa.fr

Managed Agreement: Generalizing Two Fundamental Distributed Agreement Problems

E. Anceaume R. Friedman* M. Gradinariu
* Computer Science Department, The Technion, Haifa

Systèmes communicants
Projet Adept

Publication interne n ° 1785 — february 2006 — 19 pages

Abstract: *Consensus* and *Non-Blocking Atomic Commit* (NBAC) are two fundamental distributed problems. The specifications of both problems may lead one to think that they are very similar. However, a black-box solution to any of them is not sufficient to solve the other. This paper presents a family of agreement problems called *Managed Agreement*, which is parameterized by the number of *aristocrat* nodes in the system; NBAC is a special case of this family when all nodes are aristocrats while Consensus is a special case of this family when there are no aristocrats. The paper also presents a parameterized family of failure detectors $\mathcal{F}(A)$ such that $\mathcal{F}(A)$ is the weakest failure detector class that enables solving Managed Agreement with a set A of aristocrats in an asynchronous environment.

Key-words: consensus, non-blocking atomic commit, quitable consensus, managed agreement, failure detectors

(Résumé : tsvp)

Centre National de la Recherche Scientifique
(UMR 6074) Université de Rennes 1 – Insa de Rennes
Institut National de Recherche en Informatique
et en Automatique – unité de recherche de Rennes

Accord contrôlé : génaralisation de deux problèmes d'accord fondamentaux

Résumé : Le consensus et la validation atomique non-bloquante sont deux problèmes d'accord fondamentaux. Leur spécification relativement proche peut en première approximation laisser penser que ces deux problèmes sont très similaires. Et pourtant, la solution à l'un d'entre eux ne permet pas de rśoudre l'autre.

Ce rapport technique présente une famille de problèmes d'accord, appelée par la suite *Accord contrôlé*, paramétrée par le nombre d'*aristocrates* impliqus dans le système, un aristocrate étant un nœud ayant un statut particulier dans la spécification du problème d'accord considéré. La validation atomique non-blocante est un cas particulier de cette famille lorsque tous les nœuds impliqués sont des aristocrates, alors que pour le consensus, aucun des nœuds n'a ce statut. Ce rapport technique présente également une famille de dtecteurs de défaillances $\mathcal{F}(A)$ telle que $\mathcal{F}(A)$ soit le plus faible détecteur de défaillances permettant de résoudre le problème de l'accord contrôlé en présence de A aristocrates dans un environnement asynchrone.

Mots clés : consensus, validation atomique non-blocante, consensus quitable, accord contrôlé, détecteurs de défaillances

1 Introduction

Distributed systems are composed of a collection of *processes* (also called *nodes*) that try to obtain a common goal. In many applications, this goal may involve reaching an agreement between the participants about some course of action. For example, if a file system is replicated across multiple servers, the servers need to agree on which file operations have been executed by the system and in what order, so that the system remains consistent. If the file system is not fully replicated on all servers, then an operation like moving a file from one server to another requires both servers to agree on the move in an atomic manner, in order to avoid loss of files on one hand, and duplications on the other hand.

Consensus [16] and Non-Blocking Atomic Commit (NBAC) [2, 17] are two fundamental distributed agreement problems. Intuitively, the specification of these problems assumes that each node in a distributed system starts with a given input value and the goal of each node is to decide on some output value. However, the decided values are restricted such that: all processes that do not crash eventually decide (*termination*), the value decided by all processes is the same (*agreement*), and the value decided must be related to the initial input values (*validity*). The difference between Consensus and NBAC is in their validity requirements. Specifically, Consensus only requires that a decided value is also a value that was proposed. In NBAC, it is assumed that the possible initial values are *yes* and *no* and the possible decision values are *commit* and *abort*. If the initial value of at least one node is *no*, then the decision must be *abort*. On the other hand, if the initial values of all nodes are *yes* and there are no crash failures, then the decision value must be *commit*.

Considering the distributed file system example above, when a file is replicated on all servers, these servers can use a Consensus black-box implementation in order to decide on which operations to apply and in what order. On the other hand, if the servers are not identical, then it could happen that one node may not be able to perform an operation that others can, e.g., due to physical limitations (memory and disk space) or due to some security attributes that can only be verified on that node. In these cases, it is important to give a "veto" power to such nodes in order to avoid a situation in which the decided value is one that cannot be fulfilled. In this case, NBAC is a more appropriate abstraction than Consensus.

Despite the similarity in structure of the definitions of Consensus and NBAC, in asynchronous distributed systems prone to crash failures, these are two different problems [14]. In particular, neither problem can be solved in a purely asynchronous system. However, it was shown that the minimal synchrony required to solve Consensus is strictly weaker than the one required to solve NBAC [8] (we make this statement more precise below). On the other hand, a black-box implementation of NBAC is not sufficient to solve Consensus in an otherwise asynchronous environment.

Contributions of this paper: In this paper we propose a family of problems that we call *Managed Agreement*[1], which generalizes both NBAC and Consensus. Specifically, the definition of Managed Agreement is based on the notion of *aristocrat* nodes. In Managed Agreement, there exists a value such that if any of the aristocrats proposes this value, then a corresponding value must be decided. On the other hand, if none of the aristocrats proposed the special value and none of the aristocrats failed, then any possible decision value that corresponds to a value that was proposed can be decided on. Thus, NBAC is a special case of Managed Agreement when all nodes are aristocrats, whereas Consensus is a special case of Managed Agreement when there are no aristocrats.

[1] The term Managed Agreement is inspired by the term "managed democracy".

Interestingly, the motivation for this work comes from a real system we have designed, in which a parameterized service providing Managed Agreement is a key component. Consider a large scale Internet wide system that enables an ad-hoc collection of nodes to perform transactions [1]. As the communication capabilities of various nodes are different, we do not wish to allow any node to veto a transaction, and moreover, to be in a position that a single failure (and in practice, a single connection timeout), would cause a transaction to abort. For example, consider a distributed auctioning and transaction system, in which any node can publish a multi-stage auction, e.g., to build a house. All nodes can send bids for various parts of the auction, and eventually the auctioneer runs an agreement protocol with the winners in order to sign a binding multi-party contract. In this case, we cannot force the auctioneer to agree to a contract it does not want, e.g., if none of the bids were good enough. However, once a contractor sends a bid, we want to avoid aborting the contract just because this contractor got disconnected. Thus, in this case we can run Managed Agreement in which the auctioneer (and only the auctioneer) acts as the aristocrat.

To that end, in this paper we also present a generic protocol for solving Managed Agreement, which is based on a transformation from Consensus to NBAC by Guerraoui [13]. The protocol we present utilizes any known Consensus protocol as a black-box and a new class of failure detector that we denote $?\mathcal{P}_{\mathrm{Ar}}(A)$, which is an extension of the known $?\mathcal{P}$ to detect crashes of aristocrats only ($?\mathcal{P}$ was also introduced in [13]). Finally, we introduce a failure detector class $\Psi_{\mathrm{Ar}}(A)$, again, an extension of the known class Ψ [8], and show that a corresponding family of failure detectors, denoted $\mathcal{F}(A) = (?\mathcal{P}_{\mathrm{Ar}}(A), \Psi_{\mathrm{Ar}}(A))$, is the weakest class of failure detectors that enables solving Managed Agreement for a given A.

2 Asynchronous Distributed Systems with Process Crashes

The computation model follows the one described in [4, 9]. The system consists of a finite set Π of $n > 1$ processes, namely, $\Pi = \{p_1, \ldots, p_n\}$. A process can fail by *crashing*, i.e., by prematurely halting. At most $f < n$ processes can fail by crashing. A process behaves correctly (i.e., according to its specification) until it (possibly) crashes. By definition, a *correct* process is a process that does not crash. A *faulty* process is one that is not correct. Until it (possibly) crashes, a process is *alive*.

Processes communicate and synchronize by sending and receiving messages through channels. Every pair of processes is connected by a channel. Channels are assumed to be reliable. There is no assumption about the relative speed of processes nor on message transfer delays: the system is asynchronous.

3 Managed Agreement problem

3.1 Problem Specification

In the Managed Agreement problem, the set of possible proposed values, *P-VALS*, can be different from the set of possible decided values, *D-VALS*. However, we require a one-to-one mapping $\mathcal{M}$ from the set *P-VALS* to the set *D-VALS*. In particular, for each value $pv \in$ *P-VALS* and value $dv \in$ *D-VALS* such that $dv = \mathcal{M}(pv)$, we say that dv is the *value that corresponds to* pv. Moreover, we identify one special value in *P-VALS* as the *default value* and denote it *Default*.[2] We also identify a set A of *aristocrats* among the entire set of nodes (this subset is a parameter). Managed agreement is then defined by the following properties:

[2]To clarify, *Default* is just a generic symbol; for example, in the case of NBAC, the value *no* is the *Default*.

- (Uniform) Agreement: No two processes decide differently.
- Termination: Every correct process eventually decides on some value.
- Managed-Obligation: If the decision value is $\mathcal{M}(\textit{Default})$, then either one of the aristocrats proposes *Default* or crashes.
- Managed-Justification: If the decision value v is different from $\mathcal{M}(\textit{Default})$, then v corresponds to a proposed value and all aristocrats propose a non default value.

Notice that a decision on a value other than $\mathcal{M}(\textit{Default})$ requires all aristocrats to propose a value different from *Default.* In particular, in runs in which at least one of the aristocrats have crashed even before the beginning of the protocol, the only possible decision value is $\mathcal{M}(\textit{Default})$.

3.2 Consensus, Quitable Consensus, and NBAC

The Quitable Consensus problem was introduced in [8] as an intermediate step in showing the weakest failure detector for solving NBAC. In this paper, we make a similar use of a generalized problem, and therefore we define Quitable Consensus here along with Consensus and NBAC.

In Consensus, Quitable Consensus, and NBAC, every correct process p_i *proposes* a value v_i and all correct processes have to *decide* on the same value v. This is captured by the Termination and Agreement properties as defined above [4, 9].

In addition, in Consensus, the decided value has to be one of the proposed values. More precisely, Consensus has to satisfy the following property:

- Cons-Validity: If a process decides v, then v was proposed by some process.

Quitable consensus is a weaker version of consensus where, if a failure has occurred, processes can also agree on a special value *Q*. The set of possible decided values belongs to *v,Q*:

- Quitable Cons-Validity: (a)If a process decides v, then v was proposed by some process. (b) If $v =$*Q*, then a failure has previously occurred.

On the other hand, in NBAC, the set of possible proposed values is { *yes, no* } whereas the set of possible decided values is { *commit, abort* }. These values are related by the following validity properties:[3]

- NBAC-Justification: If a process decides *commit*, then all processes propose *yes.*
- NBAC-Obligation: If all processes are correct and every process proposes *yes*, then the decision value is *commit.*

When the set of proposed values for Consensus is $\{0, 1\}$, this problem is sometimes referred to as *Binary Consensus*. However, the solvability of Consensus and Binary Consensus is the same for all PV for which $|PV| > 1$. The following observations relate Managed Agreement with Consensus and NBAC:

[3]To the best of our knowledge, the names "NBAC-Justification" and "NBAC-Obligation" were first proposed by M. Raynal for these properties.

Observation 1 (Managed Agreement and Consensus) *Consensus is equivalent to Managed Agreement with an empty set of aristocrats.*

Observation 2 (Managed Agreement and Quitable Consensus) *Quitable Consensus is equivalent to Managed Agreement when all processes are aristocrats but they are not allowed to propose Default.*

Observation 3 (Managed Agreement and NBAC) *NBAC is equivalent to Managed Agreement when all the processes are aristocrats.*

4 Solving Managed Agreement

4.1 Relevant Failure Detectors

We further enhance the environment, denoted $\mathcal{E}$, by assuming that each process has access to (one or more) *failure detector(s)* [4]. A failure detector is a module that provides each process with possibly inaccurate information about the occurrence of failures in the system. Below, we list three known types of failure detectors that enable solving Consensus and NBAC in an otherwise asynchronous distributed system.

The class quorum failure detector Σ: Specifically, Σ outputs at each process a set of processes such that any two sets intersect, and eventually every set output at correct processes consists only of correct processes. It was shown in [8] that Σ is the weakest failure detector to implement atomic registers.

The class leader failure detector Ω: The failure detector Ω outputs the id of some process at each process. There is a time after which it outputs the id of the same correct process at all correct processes [3]. It was shown in [8] that (Ω, Σ) is the weakest failure detector to solve Consensus for all environments.[4]

The class $?\mathcal{P}$: A failure detector that belongs to the class $?\mathcal{P}$ provides a boolean value to each process while maintaining the following property [13]:

- Anonymous Accuracy: The failure detector eventually returns true if and only if some process in the system has crashed.

The class Ψ: For an initial period of time, the output of Ψ at each process is *false*. Eventually Ψ behaves either like the failure detector (Ω, Σ) at all processes, or, in case a failure previously occurred, it may behave like the failure detector $?\mathcal{P}$ at all processes. The switch from *false* to $?\mathcal{P}$ is allowable only if a failure previously occurred. In [8] it is proved that $(\Psi, ?\mathcal{P})$ is the weakest failure detector to solve NBAC, while Ψ is the weakest failure detector to solve Quitable Consensus.

[4] Let us state for completeness that the class known as $\mathcal{P}^f$ [7] has been shown to be equivalent to Σ [11]. Also, each of the classes known as $\diamond\mathcal{W}$ and $\diamond\mathcal{S}$ have been shown to be equivalent to Ω [4, 6].

4.2 Some Observations Regarding Consensus, Quitable Consensus, and NBAC

As mentioned before, solving NBAC requires a strictly stronger environment, in terms of failure detection capabilities, than solving Consensus [8]. Moreover, Guerraoui et al. have shown a simple protocol that solves NBAC based on a black-box implementation of Consensus and the failure detector $?\mathcal{P}$ [13]. On the other hand, it is known that a black-box implementation of NBAC cannot be used to solve Consensus in an environment that is otherwise too weak to solve Consensus [13].

When examining the definitions of these problems, we make the following observations: First, in NBAC, there is some value (*no*) such that if anyone proposes it, then its corresponding decision value (*abort*) must be decided on (unless there is a crash) regardless of the other values proposed. This means that one value vetoes the other, while in Consensus all values have the same significance. Second, in NBAC and in Quitable Consensus, as soon as there is a single crash, it is permissible to decide (*abort* or Q) regardless of the values proposed, whereas in Consensus the decided value must always correspond to the values that were proposed. Additionally, NBAC is equivalent to the combination of Quitable Consensus and $?\mathcal{P}$ [8].

4.3 The $\mathcal{F}(\mathcal{A})$ Failure Detectors

First, we introduce the class of failure detectors $?\mathcal{P}_{\mathrm{Ar}}(A)$. Specifically, given a set of aristocrats A, a failure detector in $?\mathcal{P}_{\mathrm{Ar}}(A)$ provides a boolean value to each process while maintaining the following property:

- Aristocratic Accuracy: The failure detector eventually returns true if and only if some process in A has crashed.

Second, we can similarly extend the definition of Ψ to $\Psi_{\mathrm{Ar}}(A)$ in the obvious way. That is, $\Psi_{\mathrm{Ar}}(A)$ initially outputs *false* at all processes. Then it either behaves as (Ω, Σ), or if one of the aristocrats in A has crashed, it may behave as $?\mathcal{P}_{\mathrm{Ar}}(A)$. However, its behavior has to be consistent at all processes. Notice that when $A = \emptyset$, then $?\mathcal{P}_{\mathrm{Ar}}(A)$ always return *false* and $\Psi_{\mathrm{Ar}}(A)$ degenerates to (Ω, Σ). Finally, we define the class $\mathcal{F}(A) = (?\mathcal{P}_{\mathrm{Ar}}(A), \Psi_{\mathrm{Ar}}(A))$.

5 The Weakest Failure Detector to Solve Managed Agreement

5.1 Sufficiency: Managed Agreement Through Consensus

A generic protocol for solving Managed Agreement based on the availability of a failure detector $\mathcal{F}(A)$ appears in Figure 1. This protocol is a simple variant of the protocol of Guerraoui [13]. In our variant of the protocol, every aristocrat first sends its proposed value to all other processes. Every process then waits until either it has received the proposed values of all aristocrats, or $?\mathcal{P}_{\mathrm{Ar}}(A)$ part of its $\mathcal{F}(A)$ failure detector told it that one of the aristocrats has crashed. The latter is used to avoid blocking forever in case one of the aristocrats has crashed before sending its value to all other processes. Then, if a node received at least one *Default* value, or detected one aristocrat failure, it starts a Consensus protocol by proposing the value that corresponds to *Default* in the Managed Agreement specification. Otherwise, it starts the Consensus with a value that corresponds to its own proposed value. Yet, before starting the Consensus protocol, the process must wait until the $\Psi_{\mathrm{Ar}}(A)$ failure detector makes up its mind on whether it behaves as $?\mathcal{P}_{\mathrm{Ar}}(A)$ or as (Ω, Σ). In the former case, this means that one of the aristocrats has failed, and this has been observed by all correct processes. Thus, it is safe to decide $\mathcal{M}$(*Default*). Otherwise, the consensus is invoked, since

Function ManagedAgreement(A,v_i, k_i)

% k_i is a parameter aimed as distinguishing between different instantiations of the protocol
if I am an aristocrat (in A) **then**
send (VOTE,v_i, k_i) to everyone;
endif;
wait until either (VOTE,-,k_i) messages have been received from every aristocrat or $?\mathcal{P}_{\mathrm{Ar}}(A)$ returns true;
if received a (VOTE,*Default*,k_i) message from at least one aristocrat or $?\mathcal{P}_{\mathrm{Ar}}(A)$ returned true **then**
let $u_i := \mathcal{M}(\mathit{Default})$
else
let $u_i := \mathcal{M}(v_i)$
endif;
while ($\Psi_{\mathrm{Ar}}(A)$ = *false*) **do** nop done
if $\Psi_{\mathrm{Ar}}(A)$ = *true* **then** /* $\Psi_{\mathrm{Ar}}(A)$ behaves as ?P(A)
return $\mathcal{M}(\mathit{Default})$
else
val:=consensus(u_i, k_i); /* $\Psi_{\mathrm{Ar}}(A)$ behaves as (Ω,Σ)
return val
endif

Figure 1: A Managed Agreement Protocol Based on $\mathcal{F}(A)$ and a Consensus Subroutine

it is known that the failure detector's output obeys (Ω, Σ), and thus the Consensus protocol will terminate correctly. In particular, the use of Consensus ensures agreement between all nodes while verifying that the agreed value also maintains validity. Let us note that when there are no aristocrats, this protocol trivially degenerates to invoking Consensus with the initial values.

Lemma 1 *The protocol in Figure 1 solves the Managed Agreement problem in asynchronous environments in which processes are equipped with a failure detector from the class $\mathcal{F}(A)$ and a Consensus subroutine.*

Proof: It is easy to see that the agreement property trivially holds. If a process decides $\mathcal{M}(\mathit{Default})$ due to finding $\Psi_{\mathrm{Ar}}(A)$ = *true*, then by definition, all correct processes do the same and decide $\mathcal{M}(\mathit{Default})$. Otherwise, if Consensus is invoked, then all processes decide the output of Consensus, and thus agreement follows from the correctness of the Consensus protocol. Similarly, the termination property holds due to the termination property of the Consensus protocol and the use of the $?\mathcal{P}_{\mathrm{Ar}}(A)$ part of $\mathcal{F}(A)$ in the **wait** statement. Thus, we only need to show validity.

Clearly, if processes find $\Psi_{\mathrm{Ar}}(A)$ = *true*, then this means that one of the aristocrats has crashed. In this case, they all decide $\mathcal{M}(\mathit{Default})$ and therefore validity if preserved. Therefore, for the rest of this proof we concentrate only on runs in which the $\Psi_{\mathrm{Ar}}(A)$ part of the failure detector acts like (Ω, Σ). Let us first consider runs of the protocol in which none of the aristocrats crashes. In these runs, at the end of the **wait** statement, every alive process has all the proposed values of all the aristocrats. Thus, if any of the aristocrats has proposed the default value, then all processes (that do not crash beforehand) start the Consensus with the corresponding value u. By the validity of Consensus, the returned value by Consensus must also be u. Therefore, in these cases Managed-Obligation is observed. Alternatively, if none of the aristocrats proposes *Default*, then every process (that does not crash beforehand) starts the Consensus with a value u_i that corresponds to its proposed value v_i. By the validity of Consensus, the decided value must be one of these u_i values. Consequently, Managed-Justification is observed in these runs.

The only thing left to show now is that in runs in which at least one aristocrat proposes *Default* and at least one aristocrat crashes (either the same aristocrat or a different one), then the decided value corresponds to the default value. When considering such a run, at the end of the **wait** statement, every alive process either has received at least one *Default* value, or has had its $?\mathcal{P}_{\mathrm{Ar}}(A)$ return *true*. In either case, the process starts Consensus with $\mathcal{M}(\mathit{Default})$. By the validity of Consensus, this is also the decided value. ■

---

output$_{p_i}$:= *false*;
k_i := 1;
select $v_i \in$ *P-VALS* $\setminus \{\mathit{Default}\}$
repeat
  t_i := `ManagedAgreement`(A, v_i, k_i);
  k_i := $k_i + 1$;
until $t_i = \mathcal{M}(\mathit{Default})$;
output$_{p_i}$:= *true*;

Figure 2: From Managed Agreement to $?\mathcal{P}_{\mathrm{Ar}}(A)$

5.2 Necessity: the Minimal Failure Detector for Solving Managed-Agreement

In the following, we show that any failure detector $\mathcal{D}$ that enables solving Managed Agreement can be transformed into both $?\mathcal{P}_{\mathrm{Ar}}(A)$ and $\Psi_{\mathrm{Ar}}(A)$. Consequently, it implements $\mathcal{F}(A)$.

5.2.1 From Managed-Agreement to $?\mathcal{P}_{\mathrm{Ar}}(A)$

In the following, we show that any failure detector $\mathcal{D}$ that enables solving Managed Agreement in any environment can be transformed into $?\mathcal{P}_{\mathrm{Ar}}(A)$. The transformation algorithm that we use is similar to the one proposed in [13] for emulating $?\mathcal{P}$ from NBAC. The algorithm works as follows. Each process p_i has a local boolean variable *output*$_{p_i}$, which provides the information that should be returned by its local failure detector $?\mathcal{P}_{\mathrm{Ar}(A)}$. We assume the existence of the function `ManagedAgreement`() that solves the Managed Agreement problem. Each process p_i initiates *output*$_{p_i}$ to *false* and then repeatedly invokes the Managed Agreement function with a non default value. This is done forever, unless the returned value is $\mathcal{M}(\mathit{Default})$, in which case p_i changes *output*$_{p_i}$ to *true* and exits. The idea is that by the definition of Managed Agreement, $\mathcal{M}(\mathit{Default})$ can only be returned in this case if and only if at least one of the aristocrats has failed. The exact pseudo-code appears in Figure 2 and the proof is given below.

Lemma 2 *The transformation algorithm in Figure 2 emulates the failure detector* $?\mathcal{P}_{\mathrm{Ar}(A)}$.

Proof: We prove that the transformation algorithm verifies the Aristocratic Accuracy property, i.e., it eventually returns *true* if and only if some process in A has crashed. Suppose the transformation algorithm outputs *true* at some point. This can happen only if some invocation of the `ManagedAgreement`() function returns $\mathcal{M}(\mathit{Default})$. By definition, this can happen either if one of the aristocrats proposes *Default* or crashes. The first scenario is impossible since all processes invoke `ManagedAgreement`() with a non *Default* value. Therefore, the output can be *true* only due to an aristocrat's crash.

Let us consider now an execution of the protocol where an aristocrat crashes, and assume w.l.o.g. that it crashes before invoking the kth instance of `ManagedAgreement()`. Also, let p_i be a correct process executing the transformation algorithm. By the Termination property, the kth invocation by p_i of `ManagedAgreement()` in the transformation algorithm eventually returns; by the Managed-Obligation property, the returned value in this case must be $\mathcal{M}(\textit{Default})$. Therefore, the transformation algorithm terminates by setting the output to *true*. ■

5.2.2 From Managed Agreement to $\Psi_{\mathrm{Ar}}(A)$

The transformation from Managed Agreement to $\Psi_{\mathrm{Ar}}(A)$ is inspired by the transformation that was first proposed in [3], and then in [8], to extract Ψ from any failure detector $\mathcal{D}$ that solves Quitable Consensus. Specifically, let $\mathcal{D}$ be an arbitrary failure detector that can be used to solve Managed Agreement in some environment $\mathcal{E}$. Let $\mathcal{A}lg$ be an algorithm that uses $\mathcal{D}$ to solve Managed Agreement in $\mathcal{E}$. We must prove that $\Psi_{\mathrm{Ar}}(A)$ can be "extracted" from $\mathcal{D}$ in environment $\mathcal{E}$, i.e., processes can run in $\mathcal{E}$ a transformation algorithm that uses $\mathcal{D}$ and $\mathcal{A}lg$ to generate the output of $\Psi_{\mathrm{Ar}}(A)$— a failure detector that initially outputs $\perp$ and later behaves either like (Ω, Σ) or like $?\mathcal{P}_{\mathrm{Ar}(A)}$. The reduction algorithm $T_{\mathcal{D}\rightarrow\Psi_{\mathrm{Ar}}(A)}$ is shown in Figure 3, and is now explained. Since $\mathcal{A}lg$ solves Managed Agreement, we can assume, w.l.o.g., that it solves this problem when $\{0, 1, \textit{Default}\} \in \textit{P-VALS}$ and $\mathcal{M}(i) = i$ for $i \in \{0, 1\}$. In the construction, each process p starts by outputting $\perp$. Then, p runs two tasks in parallel whose goal is to determine whether (Ω, Σ) or $?\mathcal{P}_{\mathrm{Ar}(A)}$ has to be extracted and then perform the corresponding extraction. In the first task (Task 1), p simulates runs of $\mathcal{A}lg$ that could have occurred in the current failure pattern F and the current failure detector history of $\mathcal{D}$, exactly as in [3] (see below for extended definitions). It does this by "sampling" its local failure detector $\mathcal{D}$ and exchanging failure detector samples with the other processes (Line 5 in Figure 3). Process p organizes these samples into ever-increasing DAG G_p whose edges are consistent with the order in which the failure detectors samples were taken. Using G_p, p simulates ever-increasing partial runs of $\mathcal{A}lg$ that are compatible with paths in G_p. A path from the root of a tree to a node x in the tree corresponds to the schedule of a partial run of the algorithm, where every edge along the path corresponds to a step of some process.

Each process p organizes these runs in a forest induced by $T = \begin{pmatrix} n+p-1 \\ n \end{pmatrix}$ configurations, with n the number of processes, and p the number of different input values, i.e., $p = 3$. This forest, denoted Υ_p, contains T trees. We can order these configurations in an order that guarantees that configurations I and I^{i-1}, with $0 \leq i < T$ differ only in the value of one proposition. These trees are ordered such that Υ_p^0 corresponds to simulated runs of $\mathcal{A}lg$ in which all the processes propose 0, Υ_p^k, with some $k > 0$, corresponds to simulated runs of $\mathcal{A}lg$ in which all the processes propose 1, and Υ_p^{T-1} in which all the processes propose *Default*. Note that it exists an ordering that guarantees that there is no k' with $0 \leq k' \leq k$ such that some process proposes *Default* in $\Upsilon_p^{k'}$.

Processes periodically query their failure detectors. The results of these queries include failure and temporal information. Each process exchanges the results of its queries with all the other processes. Upon receipt of such information, a process construct a DAG [3] by incorporating the received information to its own DAG. (Each process exchanges its whole DAG with all the other processes. The temporal information enables to incorporate the received DAG with the local one.) Thus every (correct) process can construct ever-increasing finite approximations of the same infinite limit DAG G. This DAG is then used to simulate runs of managed agreement. Specifically, each path g within the DAG G can be used to simulate schedules of runs of managed agreement. That is, a path g represents several possible schedules and failure detectors

Initially:

(1) $T := \begin{pmatrix} n+p-1 \\ n \end{pmatrix}$

(2) $\Psi_{\mathrm{Ar}}(A) - output_p := \perp$ $\{\Psi_{\mathrm{Ar}}(A) - output_p$ is the output of module $\Psi_{\mathrm{Ar}}(A)$ at $p\}$

Task 1:

(3) **do forever** { same construction as in [3]}

(4) **cobegin**

(5) p builds an ever-increasing DAG G_p of failure detectors samples by repeatedly sampling its failure detector $\mathcal{D}$ and exchanging these samples with the other processes

(6) ||

(7) p uses G_p and the T initial configurations to construct a forest Υ_p of ever-increasing simulated runs of $\mathcal{A}lg$ using $\mathcal{D}$ that could have occurred with the current failure pattern F and the current failure detector history.

(8) **coend**

Task 2:

(9) **wait until** (for each tree Υ_p^i, with $1 \leq i \leq T$, p decides in one of the runs of Υ_p^i)

(10) **if** ($\exists$ i such that p decides $\mathcal{M}(\textit{Default})$ in Υ_p^i and I^i does not contain any *Default* value proposed by an aristocrat) **then**

(11) p runs $\mathcal{A}lg$ with *Default* as input value

(12) **else** {in each tree Υ_p^i, there exists a run in which the decision value is either 0, 1, or $\mathcal{M}(\textit{Default})$ but in the latter case, I^i contains the *Default* value}

(13) select two configurations I^{i-1} and I^i, with $1 \leq i \leq (T-1)$ and two schedules S and S' such that in $S(I^{i-1})$ p decides u, in $S'(I^i)$ p decides v, with $u, v \in \{0,1\}$ and $u \neq v$

(14) p runs $\mathcal{A}lg$ with (I, I', S, S') as input value

(15) **wait until** (p decides in $\mathcal{A}lg$)

(16) **if** (p decides $\mathcal{M}(\textit{Default})$) **then**

(17) $\Psi_{\mathrm{Ar}}(A) - output_p := true$ $\{\Psi_{\mathrm{Ar}}(A)$ behaves as $?\mathcal{P}_{\mathrm{Ar}(A)}$ }

(18) **else** {p decides (I_0, I_1, S_0, S_1)}

(19) $\Omega - output_p := p$

(20) $\Sigma - output_p := \Pi$

(21) **cobegin**

{ extraction of Ω }

(22) **do forever**

(23) $\Omega - output_p \leftarrow q$ such that p extracts q following the procedure in [3]

(24) ||

{ extraction of Σ }

(25) let $\mathcal{C}$ be the set of configurations reached by applying all prefixes of S_0 to I_0 and S_1 to I_1

(26) **do forever**

(27) $\Sigma - output_p \leftarrow \bigcup_{C \in \mathcal{C}}$ set of processes that p extracts following the procedure in [8]

(28) ||

(29) **do forever**

(30) $\Psi_{\mathrm{Ar}}(A) - output_p := (\Omega - output_p, \Sigma - output_p)$

(31) **coend**

(32) **endif**

(33) **endif**

Figure 3: Extraction of $\Psi_{\mathrm{Ar}}(A)$ from $\mathcal{D}$ and Managed Agreement algorithm $\mathcal{A}lg$ – code for process p

values for the processes during their execution of managed agreement. There are many different schedules that match a path in DAG G because each schedule depends on the order in which messages are received. Thus, if we consider each initial configuration I^i (that is the tuples of initial values), then one can construct a tree rooted at I^i. The set of vertices of the tree rooted at I^i is the set of all possible schedules that can occur from the given configuration I^i. An edge corresponds to an event "receipt by a process p of a message m, and the failure detector value seen by the sender of the message when it sent m to p". By considering the T different configurations, one obtains a forest of simulated runs of managed agreement. Thus the infinite limit DAG G induces an *infinite limit forest*, Υ. The *limit tree* of Υ_p^i is denoted Υ^i. Each node S of the limit forest Υ is tagged by the set of decisions that correct processes reach in the partial runs that are the descendants of S. These tags can be either univalent, i.e., 0-valent or 1-valent or $\mathcal{M}(\textit{Default})$-valent, or multi-valent (i.e., with more than one tag). We use the same definition for a critical index as in [8]: Index $i \in \{0, \ldots, T-1\}$ is critical if the root of Υ^i is multivalent or the root of Υ^i is u-valent and the root of Υ^{i-1} is v-valent, with $u, v \in \{0, 1, \mathcal{M}(\textit{Default})\}$ and $u \neq v$.

In task 2, p waits until it decides in some simulated run of every tree of the forest Υ_p (Line 9 in Figure 3). If p decides $\mathcal{M}(\textit{Default})$ in any of these runs and the initial configuration of this run does not contain any *Default* value proposed by an aristocrat then a failure must have occurred (in the current failure pattern). Note that this condition is stronger than the one in [8] because of the Managed Obligation property of Managed Agreement. Thus p knows that it is legitimate to propose the extraction of $?\mathcal{P}_{\mathrm{Ar}}(A)$. Otherwise, p's decision values in the simulated runs are 0s, 1s or $\mathcal{M}(\textit{Default})$ but in this latter case, the initial configuration contains the *Default* value (proposed by an aristocrat), and thus does not tell anything regarding failures. Thus p determines that it is possible to extract (Ω, Σ). Note that by the validity properties of Managed Agreement (Obligation and Justification), starting from an initial configuration in which there is no *Default* values (proposed by an aristocrat), the decision value may be either $\mathcal{M}(\textit{Default})$ if an aristocrat has failed after having voted and this failure has been detected before the receipt of his vote, or a decision value $v \neq \mathcal{M}(\textit{Default})$, otherwise.

At this point, p executes the given Managed Agreement algorithm $\mathcal{A}lg$ to agree with all the processes on whether to extract $?\mathcal{P}_{\mathrm{Ar}}(A)$ (because at least p has detected an aristocrat failure) or to extract (Ω, Σ). Specifically, in the former case, process p invokes an instance $\mathcal{A}lg$ of Managed Agreement by proposing *Default*. In the latter case, it invokes $\mathcal{A}lg$ with (I, I', S, S') value, where I and I' are initial configurations that differ only in the proposal of one process while S and S' are schedules in the simulated forest so that the process decides u in $S(I)$ and v in $S'(I')$, where $u, v \in \{0, 1\}$ and $u \neq v$. The existence of such configurations and schedules is shown in Lemma 3 below.

If processes decide to extract (Ω,Σ), they continue the simulation of runs of $\mathcal{A}lg$ to do this extraction. Note that the extraction of (Ω, Σ) cannot start if the decision value in every simulated run is $\mathcal{M}(\textit{Default})$. Notice that the failure of an aristocrat may not be detected. Hence the necessity of the algorithm shown in Figure 2 that emulates $?\mathcal{P}_{\mathrm{Ar}}(A)$ with Managed Agreement. If $\mathcal{A}lg$ returns $\mathcal{M}(\textit{Default})$, then the transformation algorithm starts to behave like $?\mathcal{P}_{\mathrm{Ar}}(A)$: p stops outputting $\perp$ and outputs *true* from that time on (Line 17). If $\mathcal{A}lg$ returns a value of the form (I, I_0, S, S_0), then p stops outputting $\perp$ and starts extracting Ω (Line 22) and Σ (Line 25). The extraction of Ω is done using the procedures of both [3] and [8]. To extract Ω, p must continuously output the identifier of a process such that eventually, correct processes output the identifier of the same correct process. The existence of a correct process relies on the existence of a critical index (see Lemma 4 below). Finally, the extraction of Σ is done exactly as in [8] and detailed in [15].

Lemma 3 *If any process reaches Line 12, then there are initial configurations I and I', and schedules S and S' in Υ_p, such that (a) I and I' differ in only one proposition, and (b) p decides u in $S(I)$ and p decides v in $S'(I')$, with $u, v \in \{0, 1\}$ and $u \neq v$.*

Proof: (Similar to Lemma 2 in [12]). If any process p reaches Line 12, then in each tree of Υ_p, there is a run in which p decides u with $u \in \{0, 1, \mathcal{M}(\textit{Default})\}$ (but in the latter case, some process must have proposed *Default*). By construction of the forest, Υ_p^0 corresponds to the tree in which all the processes proposed value 0, thus p's decison value must be 0. Similarly, there is a k such that in tree Υ_p^k, all the processes proposed value 1, thus p's decision value must be 1. Furthermore, there is no k' with $0 \leq k' \leq k$ such that some process proposes *Default* in $\Upsilon_p^{k'}$. The result immediately follows. ■

Lemma 4 *If any process reaches Line 22, then the limit forest Υ has a critical index.*

Proof: (Adaptation of Lemma 3 in [8]). If a process reaches Line 22, then it must have previously decided a tuple (I_0, I_1, S_0, S_1). By the Managed-Justification of Managed Agreement, some process q must have proposed this tuple to algorithm $\mathcal{A}lg$. Since q proposed this tuple, it must have decided some value v different from $\mathcal{M}(\textit{Default})$ in some run of Υ_q (this follows from the choice of tuple (I_0, I_1, S_0, S_1)). By construction of the limit forest, all the correct processes are aware of the partial run that allowed q to decide value v, and include this partial run to their own forest. By Termination and Agreement of Managed Agreement, all the correct processes decide $v \neq \mathcal{M}(\textit{Default})$ in some run of Υ.

From above, the root of some tree Υ^i is tagged with $v \neq \mathcal{M}(\textit{Default})$. Two cases are possible. Either the root is uni-valent, i.e., it it tagged with only value v, or it is multi-valent, i.e., it is tagged with both v and other tags ($1 - v$, and/or $\mathcal{M}(\textit{Default})$). In the latter case, we are done by the definition of a critical index. In the former case, all the roots are uni-valent. We consider two cases: Suppose first that $v = 1$. Two sub-cases are possible. Either ***a)*** i is less than or equal to k (recall that index k corresponds to the tree in which all the processes propose value 1), or ***b)*** i is greater than k. In sub-case ***a)***, by considering the sequence $\Upsilon^0, \ldots, \Upsilon^i, \ldots, \Upsilon^k$, there must exist some index k' with $0 < k' \leq i$ such that the root of $\Upsilon^{k'-1}$ is 0-valent while the root of $\Upsilon^{k'}$ is 1-valent. By definition k' is a critical index. In sub-case ***b)***, by considering the sequence $\Upsilon^k, \ldots, \Upsilon^i, \ldots, \Upsilon^{T-1}$, there must exist some index k'' with $i < k'' \leq T - 1$ such that the root of $\Upsilon^{k''-1}$ is 1-valent while the root of $\Upsilon^{k''}$ is u-valent, with $u \in \{0, \mathcal{M}(\textit{Default})\}$. By definition k'' is a critical index. The case for $v = 0$ is similar to the case $v = 1$. Briefly, for sub-case ***a)***, by considering the sequence $\Upsilon^0, \ldots, \Upsilon^i, \ldots, \Upsilon^k$, there must exist some index k' with $i < k' \leq i$ such that the root of $\Upsilon^{k'-1}$ is 0-valent while the root of $\Upsilon^{k'}$ is 1-valent. By definition k' is a critical index. In sub-case ***b)***, by considering the sequence $\Upsilon^k, \ldots, \Upsilon^i, \ldots, \Upsilon^{T-1}$, there must exist some index k'' with $k < k'' \leq i$ such that the root of $\Upsilon^{k''-1}$ is u-valent, with $u \in \{1, \mathcal{M}(\textit{Default})\}$ while the root of $\Upsilon^{k''}$ is 0-valent. By definition k'' is a critical index. This concludes the proof. ■

Theorem 1 *For all environments $\mathcal{E}$, if failure detector $\mathcal{D}$ can be used to solve Managed Agreement in $\mathcal{E}$, then the algorithm shown in Figure 3 transforms $\mathcal{D}$ into $\Psi_{\mathrm{Ar}}(A)$ in $\mathcal{E}$.*

Proof: The proof follows the lines of Theorem 6 in [8]. The only difference concerns the validity property. Specifically, for each process p, if $\Psi_{\mathrm{Ar}}(A) - output_p$ is true then it must be the case that some aristocrat has crashed during the current run. Suppose that process p outputs true in Line 17 in Figure 3, then p decided $\mathcal{M}(\textit{Default})$ in the current execution of algorithm $\mathcal{A}lg$ (see Line 16). Thus, by Managed-Obligation of

Managed Agreement, it must be the case that some process q invoked $\mathcal{A}lg$ with value *Default* as initial proposition (see Line 11) or that some aristocrat crashed during the current execution of $\mathcal{A}lg$. In the latter case, we are done. In the former case, q invoked $\mathcal{A}lg$ with value *Default* only if q decided $\mathcal{M}(\textit{Default})$ in one of the simulated runs of $\mathcal{A}lg$ and if the initial configuration of this run did not contain any *Default* value proposed by some aristocrat. Thus, by Managed-Obligation of Managed Agreement, it must be the case that some aristocrat crashed during this run. The rest of the proof of this theorem is exactly the same as in [8].■

Theorem 2 *$\mathcal{F}(A)$ is the weakest class of failure detectors that enables solving Managed Agreement.*

Proof: The theorem follows directly from Lemma 1, Lemma 2 and Theorem 1. ■

6 Quitable Aristocratic Consensus

6.1 The Quitable Aristocratic Agreement Problem

The proof about the weakest failure detector required to solve the NBAC problem [8] has utilized the *Quitable Consensus* (QC) intermediate problem. In this section, we extend the definition of Quitable Consensus to incorporate aristocrats, resulting in the *Quitable Aristocratic Consensus* (QAC). We then show that the following results about QAC: (1) QAC can be implemented using a $\Psi_{\mathrm{Ar}}(A)$ failure detector and Consensus protocol, (2) QAC can be used to derive a failure detector in $\Psi_{\mathrm{Ar}}(A)$, and (3) Managed Agreement can be derived from QAC and a $?\mathcal{P}_{\mathrm{Ar}}(A)$ failure detector. Clearly, (1) and (2) imply that $\Psi_{\mathrm{Ar}}(A)$ is the weakest failure detector for solving QAC. This mimics the results in [8] about QC, indicating the generality of the aristocrat nodes model.

Specifically, in the *Quitable Aristocrat Agreement* problem, denoted also QAC, *D-VALS* = *P-VALS*$\cap\{Q\}$. It is defined by the Agreement and Termination properties as defined in Section 3.1 and the following validity property:

- Quitable Aristocrat Cons-Validity: (a) If a process decides $v \in P - VALS$, then v was proposed by some process. (b) If $v =$*Q*, then a failure of an aristocrat has previously occurred.

6.2 The Weakest Failure Detector for Solving QAC

We now show (Figure 4) that QAC can be solved using a $\Psi_{\mathrm{Ar}}(A)$ failure detector assuming we have access to a protocol implementing Consensus that can be used as a subroutine. Since Consensus can be solved with any failure detector from the class (Ω, Σ), this implies that QAC can be solved with any failure detector from the class $\Psi_{\mathrm{Ar}}(A)$.

The protocol in Figure 4 is a generalization of a similar protocol for implementing Quitable Consensus from Consensus and Ψ that appeared in [8]. In this protocol, nodes wait until the output of the failure detector becomes different from *false*. Then, if it becomes *true*, this means that an aristocrat has failed, and all surviving processes will output this same value, not necessarily at the same moment. Hence, it is safe to return *Q*. Otherwise, the failure detector decided to behave like (Ω, Σ), and thus we can run Consensus, which is solvable with (Ω, Σ).

Lemma 5 *The protocol in Figure 1 solves the QAC problem in asynchronous environments in which processes are equipped with a failure detector from the class $\Psi_{\mathrm{Ar}}(A)$.*

Function QAC(A,v_i, k_i)

```
% k_i is a parameter aimed as distinguishing between different instantiations of the protocol
while Ψ_Ar(A) = false do nop done
if Ψ_Ar(A) = true then
    % Ψ_Ar(A) behaves like ?P_Ar(A) and an aristocrat has failed
    return Q
else
    val := consensus(v_i, k_i);
    % the consensus subroutine can be implemented with any (Ω, Σ) based protocol
    return val
endif
```

Figure 4: A $\Psi_{\mathrm{Ar}}(A)$-Based QAC Protocol

Proof: It is easy to see that the agreement property trivially holds due to the use of Consensus. Similarly, the termination problem holds due to the termination property of the Consensus protocol and the use of $\Psi_{\mathrm{Ar}}(A)$ in the **wait** statement. In particular, if any node starts executing the Consensus subroutine, then all alive nodes will do so. Moreover, in this case, the failure detector has decided to behave like (Ω, Σ), meaning that the Consensus protocol will terminate. Thus, we only need to show validity.

Let us first consider runs of the protocol in which none of the aristocrats crashes. In these runs, at the end of the **while** statement, all processes that have not crashed by then continue to invoke the Consensus subroutine and will return its decision value. By the validity of Consensus, the returned value must be a proposed value, and thus Quitable Aristocrat Cons-Validity is preserved.

Thus, the only thing left to show now is that in runs in which at least one aristocrat aristocrat crashes. Here, due to the properties of $\Psi_{\mathrm{Ar}}(A)$, all processes that are alive by the end of their **while** statement either execute the Consensus subroutine or return *Q*. If they all return *Q*, then since a failure of an aristocrat has occurred, then Quitable Aristocrat Cons-Validity is maintained. Otherwise, if they all execute the Consensus subroutine, then by the validity of Consensus, the value all processes return is also a value that was proposed, which is again in line with the Quitable Aristocrat Cons-Validity requirement. ■

Next, we need to show that any failure detector that enables implementing Quitable Aristocratic Consensus is at least as strong as $\Psi_{\mathrm{Ar}}(A)$. For this, we use the transformation proposed in [8] that extracts Ψ from any failure detector $\mathcal{D}$ and Quitable Consensus. In fact, the extraction protocol is verbatim the same, and the only difference is in the invocation of Quitable Aristocratic Consensus instead of Quitable Consensus.

Intuitively, the transformation algorithm executes two parallel tasks — a *simulation task* and an *extraction task*. The simulation task aims at simulating runs of Quitable Aristocratic Consensus that could have occurred in the current failure pattern with a failure detector $\mathcal{D}$. Each process organizes these runs in a forest of $n + 1$ trees. The runs in the i-th tree correspond to the simulation of Quitable Aristocratic Consensus where processes $p_1, \ldots, p_i$ propose 1 and $p_{i+1}, \ldots p_n$ propose 0. The extraction task, on the other hand, aims at deciding (based on the information provided by the simulation task) the failure detector that has to be extracted, i.e., $?\mathcal{P}_{\mathrm{Ar}}(A)$ or (Ω, Σ). A process starts this task by waiting for the decision in some run in each tree constructed by the simulation task. If the decision is *Q*, i.e., an aristocrat failure has occurred, then the process proposes the extraction of $?\mathcal{P}_{\mathrm{Ar}}(A)$. Otherwise, if every tree has a run where the decision is 0 or 1, it proposes the extraction of (Ω, Σ). More precisely, in the former case, the process invokes an instance

$\mathcal{A}$ of Quitable Aristocratic Consensus by proposing 0. In the latter case, it invokes $\mathcal{A}$ with a special non zero value, *initial*. This value has the form (I, I', S, S') where I and I' are initial configurations that differ only in the proposal of one process while S and S' are schedules in the simulated forest so that the process decides 0 in $S(I)$ and 1 in $S'(I')$. If $\mathcal{A}$ returns 0 or Q, then the transformation algorithm starts to behave like $?\mathcal{P}_{\mathrm{Ar}}(A)$. If the decided value is one of the proposed *initial* values, (I, I', S, S'), then it proceeds to the extraction of (Ω, Σ) following exactly the same procedure as proposed in [8] and using as input the set of configuration reached in $S(I)$ and $S'(I')$.

Note that the extraction of (Ω, Σ) cannot start if the decision value in every simulated run is *Q*. Moreover, the transformation algorithm behaves like $?\mathcal{P}_{\mathrm{Ar}}(A)$ if and only if the failure of an aristocrat has previously occurred.

Claim 1 *$\Psi_{\mathrm{Ar}}(A)$ is the minimal failure detector for solving Quitable Aristocratic Consensus.*

The formal proof of correctness, which is quite long, follows almost verbatim the proof in [8], with the exception of relying on Quitable Aristocratic Consensus instead of Quitable Consensus. For this reason, we skip the details here.

6.3 QAC vs. Managed Agreement

We now show that Managed Agreement is equivalent to the combination of Quitable Aristocrat Agreement with $?\mathcal{P}_{\mathrm{Ar}}(A)$. To this end, we first propose a generic protocol for solving Managed Agreement that uses any solution to QAC as a subroutine, as appears in Figure 5.

Lemma 6 *The protocol in Figure 5 solves the Managed Agreement problem in any environment $\mathcal{E}$ in which processes are equipped with a failure detector from the class $\mathcal{F}(A)$.*

Proof: It is easy to see that the agreement property trivially holds due to the use of a QAC subroutine. Similarly, the termination problem holds due to the termination property of the QAC protocol and the use of $?\mathcal{P}_{\mathrm{Ar}}(A)$ in the **wait** statement. Thus, we only need to show validity.

Let us first consider runs of the protocol in which none of the aristocrats crashes. In these runs, at the end of the **wait** statement, every alive process has all the proposed values of all the aristocrats. Thus, if any of the aristocrats has proposed the default value, then all processes (that do not crash beforehand) start the QAC protocol with the corresponding value u. By the validity of QAC, the returned value of this invocation must also be u (no aristocrat crashes during this run by the assumption of this case). Therefore, in these cases Managed-Obligation is observed. Alternatively, if none of the aristocrats proposes *Default*, then every process (that does not crash beforehand) starts the QAC subroutine with a value u_i that corresponds to its proposed value v_i. By the validity of QAC, the decided value must be one of these u_i values since no aristocrat crashes during this run (by assumption in this case). Consequently, Managed-Justification is observed in these runs.

The only thing left to show now is that in runs in which at least one aristocrat proposes *Default* and at least one aristocrat crashes (either the same aristocrat or a different one), then the decided value corresponds to the default proposed value. When considering such a run, at the end of the **wait** statement, every alive process either has received at least one *Default* value, or has had its $?\mathcal{P}_{\mathrm{Ar}}(A)$ return true. In either case, such a process invokes the QAC subroutine with a value that corresponds to *Default*. By the validity of QAC, the

decided value is either Q or $\mathcal{M}(\mathit{Default})$. Thus the decision value returned by our protocol is $\mathcal{M}(\mathit{Default})$. Therefore, in these cases Managed-Obligation is also observed.

■

Function ManagedAgreement(A, v_i, k_i)

% k_i is a parameter aimed as distinguishing between different instantiations of the protocol
if I am an aristocrat (in A) **then**
  send (VOTE,v_i,k_i) to everyone;
endif;
wait until either (VOTE,-,k_i) message has been received from every aristocrat or $?\mathcal{P}_{\mathrm{Ar}}(A)$ returns true;
if received a (VOTE,*Default*,k_i) message from at least one aristocrat or $?\mathcal{P}_{\mathrm{Ar}}(A)$ returned true **then**
  let $u_i := \mathcal{M}(\mathit{Default})$
else
  let $u_i := \mathcal{M}(v_i)$
endif;
val := QAC(A, u_i, k_i);
  % the the QAC subroutine can be implemented with a $\Psi_{\mathrm{Ar}}(A)$ based protocol
if $val = Q$ **then**
  return $\mathcal{M}(\mathit{Default})$
else
  return val

Figure 5: From QAC(A) + $?\mathcal{P}_{\mathrm{Ar}}(A)$ to Managed Agreement

---

Function QAC(A, v_i, k_i) % v_i is 1 or 0

% k_i is a parameter aimed as distinguishing between different instantiations of the protocol
send $< v_i, k_i >$ to all
d:=ManagedAgreement(A,*Yes*,k_i) % v_i use of the given Managed Agreement algorithm
if d = $\mathcal{M}(\mathit{Default})$ **then**
  return Q
else
  wait until received a value from each aristocrat $q \in A$
  return the smallest proposal value sent by an aristocrat

Figure 6: From Managed Agreement to QAC(A)

Lemma 7 *The protocol in Figure 6 transforms any protocol the solves the QAC problem into a solution for the Managed Agreement problem.*

The proof of Lemma 7 is straightforward and is thus omitted. Note that this transformation is a simple variant of the similar transformation from NBAC to QC presented in [8].

7 Related Problems

The k-TAG family of problems, which generalizes both binary Consensus and NBAC, was introduced by Charron-Bost and Fessant [5]. The specification of k-TAG includes the same termination and agreement properties as Consensus and NBAC, as well as the following validity property, called k-*Validity*: *1)* If at least k processes start with 0, then 0 is the only possible decision value, and *2)* If all processes start with 1 and at most $k-1$ failures occur, then 1 is the only possible decision value.

Other than being defined only for binary agreement problems, k-TAG is different than Managed Agreement on three accounts (all related to the fact that Managed Agreement gives special treatment to aristocrats while k-TAG has a symmetric flavor): First, in k-TAG, any subset of k processes can force 0 to be decided on, so there is no way to favor a specific subset of nodes. Second, in k-TAG, only if all k (or more) processes propose 0, then 0 has to be decided. On the other hand, in Managed Agreement even if only one aristocrat starts with *Default*, then the corresponding value has to be decided on. Third, in k-TAG, a failure of $k-1$ arbitrary nodes is required to allow deciding on a value that was not proposed. Differently, in Managed Agreement any failure of an aristocrat (and only failures of aristocrats) allow deciding on the special value.

Finally, the generic class of problems called ℓ-*veto* was introduced in [10]. A given problem is characterized as ℓ-veto if the minimal number of processes that can force a change in the allowed decision values is ℓ in fail free runs. For example, Consensus is an n-veto problem, while NBAC is 1-veto. The notion of ℓ-veto is different than Managed Agreement since ℓ-veto does not care about runs with failures and also treat all processes the same. On the other hand, Managed Agreement allows a specific subset of processes to control the decision value and also cares about the decided value in run-prone errors.

8 Discussion

In this paper we introduced the family of Managed Agreement problems, which is based on the notion of aristocrat nodes. This family generalizes both the problems of Consensus and NBAC into a single family of problems. In particular, we have shown a family of weakest failure detectors to solve Managed Agreement, and a generic protocol for solving such problems.

References

[1] E. Anceaume, R. Friedman, M. Gradinariu, and M. Roy. An Architecture for Dynamic Scalable Self-Managed Distributed Transactions. In *Proc. of the 6th IEEE International Symposium on Distributed Objects and Applications (DOA)*, 2004.

[2] P. Bernstein, V. Hadzilacos, and H. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, MA, 1987.

[3] T. Chandra, V. Hadzilacos, and S. Toueg. The Weakest Failure Detector for Solving Consensus. *Journal of the ACM*, 43(4):685–722, 1996.

[4] T. Chandra and S. Toueg. Unreliable Failure Detectors for Asynchronous Systems. *Journal of the ACM*, 43(4):685–722, 1996.

[5] B. Charron-Bost and F. Le Fessant. Validity Conditions in Agreement Problems and Time Complexity. In *Proc. 30th Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM)*, pages 196–207, 2004.

[6] F. Chu. Reducing Ω to $\diamond\mathcal{W}$. *Information Processing Letters*, 76(6):293–298, 1998.

[7] C. Delporte, H. Fauconnier, and R. Guerraoui. Failure Detection Lower Bounds on Registers and Consensus. In *Proc. of the 16th International Symposium on Distributed Computing (DISC)*, pages 237–251, 2002.

[8] C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, V. Hadzilacos, P. Kouznetsov, and S. Toueg. The Weakest Failure Detectors to Solve Certain Fundamental Problems in Distributed Computing. In *Proc. 23rd ACM Symposium on Principles of Distributed Computing (PODC)*, pages 338–346, 2004.

[9] M. Fischer, N. Lynch, and M. Patterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374–382, 1985.

[10] R. Friedman, A. Mostefaoui, and M. Raynal. The Notion of Veto Number for Distributed Agreement Problems. In *Proc. 6th International Workshop on Distributed Computing (IWDC)*, pages 315–325, 2004.

[11] R. Friedman, A. Mostefaoui, and M. Raynal. Building and Using Quorums Despite any Number of Process of Crashes. In *Proc. 5th European Dependable Computing Conference (EDCC)*, pages 2–19, 2005.

[12] R. Guerraoui, V. Hadzilacos, P. Kouznetsov, and S. Toueg. The weakest failure detectors to solve quitable consensus and non-blocking atomic commit. Technical report, LPD EPFL, 2004. Available at http://lpdwww.epfl.ch/publications/.

[13] Rachid Guerraoui. Non-Blocking Atomic Commit in Asynchronous Distributed Systems with Failure Detectors. *Distributed Computing*, 15(1):17–25, 2002.

[14] V. Hadzilacos. On the Relationship Between the Atomic Commitment and Consensus Problems. *Fault-Tolerant Distributed Computing*, pages 201–208, 1990.

[15] P. Kouznetsov. *Synchronizations using Failure Detectors*. PhD thesis, School of Computer and Communication Sciences, Ecole Polytechnique Fédérale de Lausanne, 2005.

[16] L. Pease, P. Shostak, and L. Lamport. Reaching Agreement in Presence of Faults. *Journal of the ACM*, 27(2):228–234, 1980.

[17] D. Skeen. Crash Recovery in a Distributed Database System. Memorandum No. UCB/ERL M82/45, Electronics Research Laboratory, Berkeley, 1982.